CSE 7393 Midlet Examples

Commander

```java
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Commander extends MIDlet {

    public void startApp() {
        Displayable d = new TextBox("TextBox", "Commander",
                                          20, TextField.ANY);

        Command c = new Command("Exit", Command.EXIT, 0);
        d.addCommand(c);
        d.setCommandListener(new CommandListener() {
            public void commandAction(Command c, Displayable s) {
                notifyDestroyed();
            }
        });

        Display.getDisplay(this).setCurrent(d);
    }

    public void pauseApp() {}

    public void destroyApp(boolean unconditional) {}
}
```
Two Alerts

import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class TwoAlerts extends MIDlet implements CommandListener {
    private Display mDisplay;
    private TextBox mTextBox;
    private Alert mTimedAlert;
    private Alert mModalAlert;
    private Command mAboutCommand, mGoCommand, mExitCommand;

    public TwoAlerts() {
        mAboutCommand = new Command("About", Command.SCREEN, 1);
        mGoCommand = new Command("Go", Command.SCREEN, 1);
        mExitCommand = new Command("Exit", Command.EXIT, 2);
        mTextBox = new TextBox("TwoAlerts", ",", 32, TextField.ANY);
        mTextBox.addCommand(mAboutCommand);
        mTextBox.addCommand(mGoCommand);
        mTextBox.addCommand(mExitCommand);
        mTextBox.setCommandListener(this);
        mTimedAlert = new Alert("Network error",
                                "A network error occurred. Please try again.",
                                null,
                                AlertType.INFO);
        mModalAlert = new Alert("About TwoAlerts",
                                "TwoAlerts is a simple MIDlet that demonstrates the use of Alerts.",
                                null,
                                AlertType.INFO);
        mModalAlert.setTimeout(Alert.FOREVER);
    }

    public void startApp() {
        mDisplay = Display.getDisplay(this);
        mDisplay.setCurrent(mTextBox);
    }

    public void pauseApp() {
    }

    public void destroyApp(boolean unconditional) {
    }

    public void commandAction(Command c, Displayable s) {
        if (c == mAboutCommand)
            mDisplay.setCurrent(mModalAlert);
        else if (c == mGoCommand)
            mDisplay.setCurrent(mTimedAlert, mTextBox);
        else if (c == mExitCommand)
            notifyDestroyed();
    }
}
public class GaugeMIDlet extends MIDlet implements CommandListener {
    private Display mDisplay;
    private Form mGaugeForm;
    private Command mUpdateCommand, mIdleCommand;
    private Gauge mInteractiveGauge;
    private Gauge mIncrementalGauge;
    private Gauge mContinuousGauge;

    public GaugeMIDlet() {
        mGaugeForm = new Form("Gauges");
        mInteractiveGauge = new Gauge("Interactive", true, 5, 2);
        mInteractiveGauge.setLayout(Item.LAYOUT_2);
        mGaugeForm.append(mInteractiveGauge);
        mContinuousGauge = new Gauge("Non-I continuous", false,
            Gauge.INDEFINITE, Gauge.CONTINUOUS_RUNNING);
        mContinuousGauge.setLayout(Item.LAYOUT_2);
        mGaugeForm.append(mContinuousGauge);
        mIncrementalGauge = new Gauge("Non-I incremental", false,
            Gauge.INDEFINITE, Gauge.INCREMENTAL_UPDATING);
        mIncrementalGauge.setLayout(Item.LAYOUT_2);
        mGaugeForm.append(mIncrementalGauge);
        mUpdateCommand = new Command("Update", Command.SCREEN, 0);
        mIdleCommand = new Command("Idle", Command.SCREEN, 0);
        Command exitCommand = new Command("Exit", Command.EXIT, 0);
        mGaugeForm.addCommand(mUpdateCommand);
        mGaugeForm.addCommand(mIdleCommand);
        mGaugeForm.addCommand(exitCommand);
        mGaugeForm.setCommandListener(this);
    }

    public void startApp() {
        if (mDisplay == null) mDisplay = Display.getDisplay(this);
        mDisplay.setCurrent(mGaugeForm);
    }

    public void pauseApp() {}

    public void destroyApp(boolean unconditional) {}

    public void commandAction(Command c, Displayable s) {
        if (c.getCommandType() == Command.EXIT)
            notifyDestroyed();
        else if (c == mUpdateCommand) {
            mContinuousGauge.setValue(Gauge.CONTINUOUS_RUNNING);
            mIncrementalGauge.setValue(Gauge.INCREMENTAL_UPDATING);
        }
        else if (c == mIdleCommand) {
            mContinuousGauge.setValue(Gauge.CONTINUOUS_IDLE);
            mIncrementalGauge.setValue(Gauge.INCREMENTAL_IDLE);
        }
    }
}
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class GaugeTracker extends MIDlet
    implements ItemStateListener, CommandListener {
private Gauge mGauge;
private StringItem mStringItem;

public GaugeTracker() {
    int initialValue = 3;
    mGauge = new Gauge("GaugeTitle", true, 5, initialValue);
    mStringItem = new StringItem(null, "[value]");
    itemStateChanged(mGauge);
}

public void itemStateChanged(Item item) {
    if (item == mGauge)
        mStringItem.setText("Value = " + mGauge.getValue());
}

public void commandAction(Command c, Displayable s) {
    if (c.getCommandType() == Command.EXIT)
        notifyDestroyed();
}

public void startApp() {
    Form form = new Form("GaugeTracker");
    form.addCommand(new Command("Exit", Command.EXIT, 0));
    form.setCommandListener(this);
    // Now add the selected items.
    form.append(mGauge);
    form.append(mStringItem);
    form.setItemStateListener(this);
    Display.getDisplay(this).setCurrent(form);
}

public void pauseApp() {}
public void destroyApp(boolean unconditional) {}

import java.io.*;
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class TravelList extends MIDlet
    implements CommandListener {
private List mList;
private Command mExitCommand, mNextCommand;

public TravelList() {
    String[] stringElements = { "Airplane", "Car", "Hotel" };
Image[] imageElements = { loadImage("/airplane.png"),
    loadImage("/car.png"), loadImage("/hotel.png") };  
mList = new List("Reservation type", List.IMPLICIT,
    stringElements, imageElements);  
mNextCommand = new Command("Next", Command.SCREEN, 0);
mExitCommand = new Command("Exit", Command.EXIT, 0);
mList.addCommand(mNextCommand);
mList.addCommand(mExitCommand);
mList.setCommandListener(this);  }

public void startApp() {
    Display.getDisplay(this).setCurrent(mList);
}

public void commandAction(Command c, Displayable s) {
    if (c == mNextCommand || c == List.SELECT_COMMAND) {
        int index = mList.getSelectedIndex();
        Alert alert = new Alert("Your selection",
            "You chose " + mList.getString(index) + ".",
            null, AlertType.INFO);
        Display.getDisplay(this).setCurrent(alert, mList);
    } else if (c == mExitCommand)
        notifyDestroyed();
}

public void pauseApp() {}

public void destroyApp(boolean unconditional) {}

private Image loadImage(String name) {
    Image image = null;
    try {
        image = Image.createImage(name);
    }
    catch (IOException ioe) {
        System.out.println(ioe);
    }
    return image;
}
public class Ticker extends Object

Implements a "ticker-tape", a piece of text that runs continuously across the display. The direction and speed of scrolling are determined by the implementation. While animating, the ticker string scrolls continuously. That is, when the string finishes scrolling off the display, the ticker starts over at the beginning of the string.

There is no API provided for starting and stopping the ticker. The application model is that the ticker is always scrolling continuously. However, the implementation is allowed to pause the scrolling for power consumption purposes, for example, if the user doesn't interact with the device for a certain period of time. The implementation should resume scrolling the ticker when the user interacts with the device again.

The text of the ticker may contain line breaks. The complete text MUST be displayed in the ticker; line break characters should not be displayed but may be used as separators.

The same ticker may be shared by several Displayable objects ("screens"). This can be accomplished by calling setTicker() on each of them. Typical usage is for an application to place the same ticker on all of its screens. When the application switches between two screens that have the same ticker, a desirable effect is for the ticker to be displayed at the same location on the display and to continue scrolling its contents at the same position. This gives the illusion of the ticker being attached to the display instead of to each screen.

An alternative usage model is for the application to use different tickers on different sets of screens or even a different one on each screen. The ticker is an attribute of the Displayable class so that applications may implement this model without having to update the ticker to be displayed as the user switches among screens.

Since:
MIDP 1.0

**Constructor Summary**

<table>
<thead>
<tr>
<th>Ticker(String str)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructs a new Ticker object, given its initial contents string.</td>
</tr>
</tbody>
</table>
Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String getString()</td>
<td>Gets the string currently being scrolled by the ticker.</td>
</tr>
<tr>
<td>void setString(String str)</td>
<td>Sets the string to be displayed by this ticker.</td>
</tr>
</tbody>
</table>

Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Ticker

public Ticker(String str)

Constructs a new Ticker object, given its initial contents string.

Parameters:

str - string to be set for the Ticker

Throws:

NullPointerException - if str is null

Method Detail

setString

public void setString(String str)

Sets the string to be displayed by this ticker. If this ticker is active and is on the display, it immediately begins showing the new string.

Parameters:

str - string to be set for the Ticker

Throws:

NullPointerException - if str is null

See Also:

getString()

getString

public String getString()

Gets the string currently being scrolled by the ticker.

Returns:

string of the ticker
See Also:
setString(java.lang.String)

javax.microedition.lcdui

Class TextBox

javax.microedition.lcdui
java.lang.Object
  javax.microedition.lcdui.Displayable
    javax.microedition.lcdui.Screen
      javax.microedition.lcdui.TextBox

public class TextBox
extends Screen

The TextBox class is a Screen that allows the user to enter and edit text.

A TextBox has a maximum size, which is the maximum number of characters that can be stored in the object at any time (its capacity). This limit is enforced when the TextBox instance is constructed, when the user is editing text within the TextBox, as well as when the application program calls methods on the TextBox that modify its contents. The maximum size is the maximum stored capacity and is unrelated to the number of characters that may be displayed at any given time. The number of characters displayed and their arrangement into rows and columns are determined by the device.

The implementation may place a boundary on the maximum size, and the maximum size actually assigned may be smaller than the application had requested. The value actually assigned will be reflected in the value returned by getMaxSize(). A defensively-written application should compare this value to the maximum size requested and be prepared to handle cases where they differ.

The text contained within a TextBox may be more than can be displayed at one time. If this is the case, the implementation will let the user scroll to view and edit any part of the text. This scrolling occurs transparently to the application.

If the constraints are set to TextField.ANY The text may contain line breaks. The display of the text must break accordingly and the user must be able to enter line break characters.

TextBox has the concept of input constraints that is identical to TextField. The constraints parameters of methods within the TextBox class use constants defined in the TextField class. See the description of input constraints in the TextField class for the definition of these constants. TextBox also has the same notions as TextField of the actual contents and the displayed contents, described in the same section.

TextBox also has the concept of input modes that is identical to TextField. See the description of input modes in the TextField class for more details.
Since:
MIDP 1.0

Constructor Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextBox(String title, String text, int maxSize, int constraints)</td>
<td>Creates a new TextBox object with the given title string, initial contents, maximum size in characters, and constraints.</td>
</tr>
</tbody>
</table>

Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void delete(int offset, int length)</td>
<td>Deletes characters from the TextBox.</td>
</tr>
<tr>
<td>int getCaretPosition()</td>
<td>Gets the current input position.</td>
</tr>
<tr>
<td>int getChars(char[] data)</td>
<td>Copies the contents of the TextBox into a character array starting at index zero.</td>
</tr>
<tr>
<td>int getConstraints()</td>
<td>Gets the current input constraints of the TextBox.</td>
</tr>
<tr>
<td>int getMaxSize()</td>
<td>Returns the maximum size (number of characters) that can be stored in this TextBox.</td>
</tr>
<tr>
<td>String getString()</td>
<td>Gets the contents of the TextBox as a string value.</td>
</tr>
<tr>
<td>void insert(char[] data, int offset, int length, int position)</td>
<td>Inserts a subrange of an array of characters into the contents of the TextBox.</td>
</tr>
<tr>
<td>void insert(String src, int position)</td>
<td>Inserts a string into the contents of the TextBox.</td>
</tr>
<tr>
<td>void setChars(char[] data, int offset, int length)</td>
<td>Sets the contents of the TextBox from a character array, replacing the previous contents.</td>
</tr>
<tr>
<td>void setConstraints(int constraints)</td>
<td>Sets the input constraints of the TextBox.</td>
</tr>
<tr>
<td>void setInitialInputMode(String characterSubset)</td>
<td>Sets a hint to the implementation as to the input mode that should be used when the user initiates editing of this TextBox.</td>
</tr>
<tr>
<td>int setMaxSize(int maxSize)</td>
<td>Sets the maximum size (number of characters) that can be contained in this TextBox.</td>
</tr>
<tr>
<td>void setString(String text)</td>
<td>Sets the contents of the TextBox as a string value, replacing the previous contents.</td>
</tr>
</tbody>
</table>
void setTicker(Ticker ticker)

Sets a ticker for use with this Displayable, replacing any previous ticker.

void setTitle(String s)

Sets the title of the Displayable.

int size()

Gets the number of characters that are currently stored in this TextBox.

Methods inherited from class javax.microedition.lcdui.Displayable

addCommand, getHeight, getTicker, getTitle, getWidth, isShown, removeCommand, setCommandListener, sizeChanged

Methods inherited from class java.lang.Object
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

TextBox

public TextBox(String title,
    String text,
    int maxSize,
    int constraints)

Creates a new TextBox object with the given title string, initial contents, maximum size in characters, and constraints. If the text parameter is null, the TextBox is created empty. The maxSize parameter must be greater than zero. An IllegalArgumentException is thrown if the length of the initial contents string exceeds maxSize. However, the implementation may assign a maximum size smaller than the application had requested. If this occurs, and if the length of the contents exceeds the newly assigned maximum size, the contents are truncated from the end in order to fit, and no exception is thrown.

Parameters:
title - the title text to be shown with the display
text - the initial contents of the text editing area, null may be used to indicate no initial content
maxSize - the maximum capacity in characters. The implementation may limit boundary maximum capacity and the actually assigned capacity may me smaller than requested. A defensive application will test the actually given capacity with getMaxSize().
constraints - see input constraints
Throws:

- `IllegalArgumentException` - if `maxSize` is zero or less
- `IllegalArgumentException` - if the `constraints` parameter is invalid
- `IllegalArgumentException` - if `text` is illegal for the specified constraints
- `IllegalArgumentException` - if the length of the string exceeds the requested maximum capacity

### javax.microedition.lcdui

#### Class List

```java
public class List
    extends Screen
    implements Choice
```

A `Screen` containing list of choices. Most of its behavior is common with class `ChoiceGroup`, and their common API. The different `List` types in particular, are defined in interface `Choice`. When a `List` is present on the display, the user can interact with it by selecting elements and possibly by traversing and scrolling among them. Traversing and scrolling operations do not cause application-visible events. The system notifies the application only when a `Command` is invoked by notifying its `CommandListener`. The `List` class also supports a select command that may be invoked specially depending upon the capabilities of the device.

The notion of a select operation on a `List` element is central to the user's interaction with the `List`. On devices that have a dedicated hardware "select" or "go" key, the select operation is implemented with that key. Devices that do not have a dedicated key must provide another means to do the select operation, for example, using a soft key. The behavior of the select operation within the different types of lists is described in the following sections.

`List` objects may be created with `Choice` types of `Choice.EXCLUSIVE`, `Choice.MULTIPLE`, and `Choice.IMPLICIT`. The `Choice` type `Choice.POPUP` is not allowed on `List` objects.

#### Selection in exclusive and multiple lists

The select operation is not associated with a `Command` object, so the application has no means of setting a label for it or being notified when the operation is performed. In `Lists` of type `EXCLUSIVE`, the select operation selects the target element and deselects the previously selected element. In `Lists` of type `MULTIPLE`, the select operation toggles the selected state of the target element, leaving the selected state of other elements
unchanged. Devices that implement the select operation using a soft key will need to provide a label for it. The label should be something similar to "Select" for Lists of type EXCLUSIVE, and it should be something similar to "Mark" or "Unmark" for Lists of type MULTIPLE.

**Selection in IMPLICIT Lists**

The select operation is associated with a Command object referred to as the *select command*. When the user performs the select operation, the system will invoke the select command by notifying the List's CommandListener. The default select command is the system-provided command SELECT_COMMAND. The select command may be modified by the application through use of the setSelectCommand method. Devices that implement the select operation using a soft key will use the label from the select command. If the select command is SELECT_COMMAND, the device may choose to provide its own label instead of using the label attribute of SELECT_COMMAND. Applications should generally provide their own select command to replace SELECT_COMMAND. This allows applications to provide a meaningful label, instead of relying on the one provided by the system for SELECT_COMMAND. The implementation must not invoke the select command if there are no elements in the List, because if the List is empty the selection does not exist. In this case the implementation should remove or disable the select command if it would appear explicitly on a soft button or in a menu. Other commands can be invoked normally when the List is empty.

**Use of IMPLICIT Lists**

IMPLICIT Lists can be used to construct menus by providing operations as List elements. The application provides a Command that is used to select a List element and then defines this Command to be used as the select command. The application must also register a CommandListener that is called when the user selects or activates the Command:

```java
String[] elements = { ... }; //Menu items as List elements
List menuList = new List("Menu", List.IMPLICIT, elements, null);
Command selectCommand = new Command("Open", Command.ITEM, 1);
menuList.setSelectCommand(selectCommand);
menuList.setCommandListener(...);
```

The listener can query the List to determine which element is selected and then perform the corresponding action. Note that setting a command as the select command adds it to the List as a side effect.

The select command should be considered as a *default operation* that takes place when a select key is pressed. For example, a List displaying email headers might have three operations: read, reply, and delete. Read is considered to be the default operation.

```java
List list = new List("Email", List.IMPLICIT, headers);
readCommand = new Command("Read", Command.ITEM, 1);
replyCommand = new Command("Reply", Command.ITEM, 2);
```
deleteCommand = new Command("Delete", Command.ITEM, 3);
list.setSelectCommand(readCommand);
list.addCommand(replyCommand);
list.addCommand(deleteCommand);
list.setCommandListener(...);

On a device with a dedicated select key, pressing this key will invoke readCommand. On a
device without a select key, the user is still able to invoke the read command, since it is
also provided as an ordinary Command.

It should be noted that this kind of default operation must be used carefully, and the
usability of the resulting user interface must always kept in mind. The default operation
should always be the most intuitive operation on a particular List.

Since:
MIDP 1.0

---

### Field Summary

| static Command SELECT_COMMAND | The default select command for IMPLICIT Lists. |

---

### Fields inherited from interface javax.microedition.lcdui.Choice

EXCLUSIVE, IMPLICIT, MULTIPLE, POPUP, TEXT_WRAP_DEFAULT, TEXT_WRAP_OFF, TEXT_WRAP_ON

---

### Constructor Summary

**List**(String title, int listType)

Creates a new, empty List, specifying its title and the type of the list.

**List**(String title, int listType, String[] stringElements, Image[] imageElements)

Creates a new List, specifying its title, the type of the List, and an array of
Strings and Images to be used as its initial contents.

---

### Method Summary

: int append(String stringPart, Image imagePart)

Appends an element to the List.

void delete(int elementNum)

Deletes the element referenced by elementNum.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>deleteAll()</code></td>
<td>Deletes all elements from this List.</td>
</tr>
<tr>
<td><code>getFitPolicy()</code></td>
<td>Gets the application's preferred policy for fitting Choice element contents to the available screen space.</td>
</tr>
<tr>
<td><code>getFont(int elementNum)</code></td>
<td>Gets the application's preferred font for rendering the specified element of this Choice.</td>
</tr>
<tr>
<td><code>getImage(int elementNum)</code></td>
<td>Gets the Image part of the element referenced by <code>elementNum</code>.</td>
</tr>
<tr>
<td><code>getSelectedFlags(boolean[] selectedArray_return)</code></td>
<td>Queries the state of a List and returns the state of all elements in the boolean array <code>selectedArray_return</code>.</td>
</tr>
<tr>
<td><code>getSelectedIndex()</code></td>
<td>Returns the index number of an element in the List that is selected.</td>
</tr>
<tr>
<td><code>getString(int elementNum)</code></td>
<td>Gets the String part of the element referenced by <code>elementNum</code>.</td>
</tr>
<tr>
<td><code>insert(int elementNum, String stringPart, Image imagePart)</code></td>
<td>Inserts an element into the List just prior to the element specified.</td>
</tr>
<tr>
<td><code>isSelected(int elementNum)</code></td>
<td>Gets a boolean value indicating whether this element is selected.</td>
</tr>
<tr>
<td><code>removeCommand(Command cmd)</code></td>
<td>The same as <code>Displayable.removeCommand</code> but with the following additional semantics.</td>
</tr>
<tr>
<td><code>set(int elementNum, String stringPart, Image imagePart)</code></td>
<td>Sets the String and Image parts of the element referenced by <code>elementNum</code>, replacing the previous contents of the element.</td>
</tr>
<tr>
<td><code>setFitPolicy(int fitPolicy)</code></td>
<td>Sets the application's preferred policy for fitting Choice element contents to the available screen space.</td>
</tr>
<tr>
<td><code>setFont(int elementNum, Font font)</code></td>
<td>Sets the application's preferred font for rendering the specified element of this Choice.</td>
</tr>
<tr>
<td><code>setSelectCommand(Command command)</code></td>
<td>Sets the Command to be used for an IMPLICIT List selection action.</td>
</tr>
<tr>
<td><code>setSelectedFlags(boolean[] selectedArray)</code></td>
<td>Sets the selected state of all elements of the List.</td>
</tr>
<tr>
<td><code>setSelectedIndex(int elementNum, boolean selected)</code></td>
<td>Sets the selected state of an element.</td>
</tr>
<tr>
<td><code>setTicker(Ticker ticker)</code></td>
<td>Sets a ticker for use with this Displayable, replacing any previous ticker.</td>
</tr>
<tr>
<td><code>setTitle(String s)</code></td>
<td>Sets the title of the Displayable.</td>
</tr>
</tbody>
</table>
size() Gets the number of elements in the List.

Methods inherited from class javax.microedition.lcdui.Displayable
addCommand, getHeight, getTicker, getTitle, getWidth, isShown, setCommandListener, sizeChanged

Methods inherited from class java.lang.Object
equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

SELECT_COMMAND

public static final Command SELECT_COMMAND
The default select command for IMPLICIT Lists. Applications using an IMPLICIT List should set their own select command using setSelectCommand.

The field values of SELECT_COMMAND are:
- label = "" (an empty string)
- type = SCREEN
- priority = 0

(It would be more appropriate if the type were ITEM, but the type of SCREEN is retained for historical purposes.)

The application should not use these values for recognizing the SELECT_COMMAND. Instead, object identities of the Command and Displayable (List) should be used.

SELECT_COMMAND is treated as an ordinary Command if it is used with other Displayable types.

Constructor Detail

List

public List(String title, int listType)
Creates a new, empty List, specifying its title and the type of the list.
Parameters:
- **title** - the screen's title (see `Displayable`)
- **listType** - one of IMPLICIT, EXCLUSIVE, or MULTIPLE

Throws:
- `IllegalArgumentException` - if `listType` is not one of IMPLICIT, EXCLUSIVE, or MULTIPLE

See Also:
- `Choice`

### List

```java
public List<String> title,
        int listType,
        String[] stringElements,
        Image[] imageElements)
```

Creates a new `List`, specifying its title, the type of the `List`, and an array of Strings and Images to be used as its initial contents.

The `stringElements` array must be non-null and every array element must also be non-null. The length of the `stringElements` array determines the number of elements in the `List`. The `imageElements` array may be null to indicate that the `List` elements have no images. If the `imageElements` array is non-null, it must be the same length as the `stringElements` array. Individual elements of the `imageElements` array may be null in order to indicate the absence of an image for the corresponding `List` element. Non-null elements of the `imageElements` array may refer to mutable or immutable images.

Parameters:
- **title** - the screen's title (see `Displayable`)
- **listType** - one of IMPLICIT, EXCLUSIVE, or MULTIPLE
- **stringElements** - set of strings specifying the string parts of the `List` elements
- **imageElements** - set of images specifying the image parts of the `List` elements

Throws:
- `NullPointerException` - if `stringElements` is null
- `NullPointerException` - if the `stringElements` array contains any null elements
- `IllegalArgumentException` - if the `imageElements` array is non-null and has a different length from the `stringElements` array
- `IllegalArgumentException` - if `listType` is not one of IMPLICIT, EXCLUSIVE, or MULTIPLE

See Also:
- `Choice.EXCLUSIVE`, `Choice.MULTIPLE`, `Choice.IMPLICIT`
public class Form
extends Screen

A Form is a Screen that contains an arbitrary mixture of items: images, read-only text fields, editable text fields, editable date fields, gauges, choice groups, and custom items. In general, any subclass of the Item class may be contained within a form. The implementation handles layout, traversal, and scrolling. The entire contents of the Form scrolls together.

**Item Management**

The items contained within a Form may be edited using append, delete, insert, and set methods. Items within a Form are referred to by their indexes, which are consecutive integers in the range from zero to size()-1, with zero referring to the first item and size()-1 to the last item.

An item may be placed within at most one Form. If the application attempts to place an item into a Form, and the item is already owned by this or another Form, an IllegalStateException is thrown. The application must remove the item from its currently containing Form before inserting it into the new Form.

If the Form is visible on the display when changes to its contents are requested by the application, updates to the display take place as soon as it is feasible for the implementation to do so. Applications need not take any special action to refresh a Form's display after its contents have been modified.

**Layout**

Layout policy in Form is organized around rows. Rows are typically related to the width of the screen, respective of margins, scroll bars, and such. All rows in a particular Form will have the same width. Rows do not vary in width based on the Items contained within the Form, although they may all change width in certain circumstances, such as when a scroll bar needs to be added or removed. Forms generally do not scroll horizontally.

Forms grow vertically and scroll vertically as necessary. The height of a Form varies depending upon the number of rows and the height of each row. The height of each row is determined by the items that are positioned on that row. Rows need not all have the same height. Implementations may also vary row heights to provide proper padding or vertical alignment of Item labels.
An implementation may choose to lay out Items in a left-to-right or right-to-left direction depending upon the language conventions in use. The same choice of layout direction must apply to all rows within a particular Form.

Prior to the start of the layout algorithm, the Form is considered to have one empty row at the top. The layout algorithm considers each Item in turn, starting at Item zero and proceeding in order through each Item until the last Item in the Form has been processed. If the layout direction (as described above) is left-to-right, the beginning of the row is the left edge of the Form. If the layout direction is right-to-left, the beginning of the row is the right edge of the Form. Items are laid out at the beginning of each row, proceeding across each row in the chosen layout direction, packing as many Items onto each row as will fit, unless a condition occurs that causes the packing of a row to be terminated early. A new row is then added, and Items are packed onto it as described above. Items are packed onto rows, and new rows are added below existing rows as necessary until all Items have been processed by the layout algorithm.

The layout algorithm has a concept of a current alignment. It can have the value LAYOUT_LEFT, LAYOUT_CENTER, or LAYOUT_RIGHT. The value of the current alignment at the start of the layout algorithm depends upon the layout direction in effect for this Form. If the layout direction is left-to-right, the initial alignment value must be LAYOUT_LEFT. If the layout direction is right-to-left, the initial alignment value must be LAYOUT_RIGHT. The current alignment changes when the layout algorithm encounters an Item that has one of the layout directives LAYOUT_LEFT, LAYOUT_CENTER, or LAYOUT_RIGHT. If none of these directives is present on an Item, the current layout directive does not change. This rule has the effect of grouping the contents of the Form into sequences of consecutive Items sharing an alignment value. The alignment value of each Item is maintained internally to the Form and does not affect the Items' layout value as reported by the Item.getLayout method.

The layout algorithm generally attempts to place an item on the same row as the previous item, unless certain conditions occur that cause a "row break." When there is a row break, the current item will be placed at the beginning of a new row instead of being placed after the previous item, even if there is room.

A row break occurs before an item if any of the following conditions occurs:

- the previous item has a row break after it;
- it has the LAYOUT_NEWLINE_BEFORE directive; or
- it is a StringItem whose contents starts with "n";
- it is a ChoiceGroup, DateField, Gauge, or a TextField, and the LAYOUT_2 directive is not set; or
- this Item has a LAYOUT_LEFT, LAYOUT_CENTER, or LAYOUT_RIGHT directive that differs from the Form's current alignment.

A row break occurs after an item if any of the following conditions occurs:

- it is a StringItem whose contents ends with "n"; or
- it has the LAYOUT_NEWLINE_AFTER directive; or
• It is a ChoiceGroup, DateField, Gauge, or a TextField, and the LAYOUT_2 directive is not set.

The presence of the LAYOUT_NEWLINE_BEFORE or LAYOUT_NEWLINE_AFTER directive does not cause an additional row break if there is one already present. For example, if a LAYOUT_NEWLINE_BEFORE directive appears on a StringItem whose contents starts with "\n", there is only a single row break. A similar rule applies with a trailing "\n" and LAYOUT_NEWLINE_AFTER. Also, there is only a single row break if an item has the LAYOUT_NEWLINE_AFTER directive and the next item has the LAYOUT_NEWLINE_BEFORE directive. However, the presence of consecutive "\n" characters, either within a single StringItem or in adjacent StringItems, will cause as many row breaks as there are "\n" characters. This will cause empty rows to be present. The height of an empty row is determined by the prevailing font height of the StringItem within which the "\n" that ends the row occurs.

Implementations may provide additional conditions under which a row break occurs. For example, an implementation's layout policy may lay out labels specially, implicitly causing a break before every Item that has a label. Or, as another example, a particular implementation's user interface style may dictate that a DateField item always appears on a row by itself. In this case, this implementation may cause row breaks to occur both before and after every DateField item.

Given two items with adjacent Form indexes, if none of the specified or implementation-specific conditions for a row break between them occurs, and if space permits, these items should be placed on the same row.

When packing Items onto a row, the width of the item is compared with the remaining space on the row. For this purpose, the width used is the Item's preferred width, unless the Item has the LAYOUT_SHRINK directive, in which case the Item's minimum width is used. If the Item is too wide to fit in the space remaining on the row, the row is considered to be full, a new row is added beneath this one, and the Item is laid out on this new row.

Once the contents of a row have been determined, the space available on the row is distributed by expanding items and by adding space between items. If any items on this row have the LAYOUT_SHRINK directive (that is, they are shrinkable), space is first distributed to these items. Space is distributed to each of these items proportionally to the difference between the each Item's preferred size and its minimum size. At this stage, no shrinkable item is expanded beyond its preferred width.

For example, consider a row that has 30 pixels of space available and that has two shrinkable items A and B. Item A's preferred size is 15 and its minimum size is 10. Item B's preferred size is 30 and its minimum size is 20. The difference between A's preferred and minimum size is 5, and B's difference is 10. The 30 pixels are distributed to these items proportionally to these differences. Therefore, 10 pixels are distributed to item A and 20 pixels to item B.

If after expanding all the shrinkable items to their preferred widths, there is still space left on the row, this remaining space is distributed equally among the Items that have the
**LAYOUT_EXPAND** directive (the stretchable Items). The presence of any stretchable items on a row will cause the Items on this row to occupy the full width of the row.

If there are no stretchable items on this row, and there is still space available on this row, the Items are packed as tightly as possible and are placed on the row according to the alignment value shared by the Items on this row. (Since changing the current alignment causes a row break, all Items on the same row must share the same alignment value.) If the alignment value is **LAYOUT_LEFT**, the Items are positioned at the left end of the row and the remaining space is placed at the right end of the row. If the alignment value is **LAYOUT_RIGHT**, the Items are positioned at the right end of the row and the remaining space is placed at the left end of the row. If the alignment value is **LAYOUTCENTER**, the Items are positioned in the middle of the row such that the remaining space on the row is divided evenly between the left and right ends of the row.

Given the set of items on a particular row, the heights of these Items are inspected. For each Item, the height that is used is the preferred height, unless the Item has the **LAYOUT_VSHRINK** directive, in which case the Item's minimum height is used. The height of the tallest Item determines the height of the row. Items that have the **LAYOUT_VSHRINK** directive are expanded to their preferred height or to the height of the row, whichever is smaller. Items that are still shorter than the row height and that have the **LAYOUT_VEXPAND** directive will expand to the height of the row. The **LAYOUT_VEXPAND** directive on an item will never increase the height of a row.

Remaining Items shorter than the row height will be positioned vertically within the row using the **LAYOUT_TOP**, **LAYOUT_BOTTOM**, and **LAYOUT_VCENTER** directives. If no vertical layout directive is specified, the item must be aligned along the bottom of the row.

**StringItems** are treated specially in the above algorithm. If the contents of a StringItem (its string value, exclusive of its label) contain a newline character ("\n"), the string should be split at that point and the remainder laid out starting on the next row.

If one or both dimensions of the preferred size of a StringItem have been locked, the StringItem is wrapped to fit that width and height and is treated as a rectangle whose minimum and preferred width and height are the width and height of this rectangle. In this case, the **LAYOUT_SHRINK**, **LAYOUT_EXPAND**, and **LAYOUT_VEXPAND** directives are ignored.

If both dimensions of the preferred size of a StringItem are unlocked, the text from the StringItem may be wrapped across multiple rows. At the point in the layout algorithm where the width of the Item is compared to the remaining space on the row, as much text is taken from the beginning of the StringItem as will fit onto the current row. The contents of this row are then positioned according to the current alignment value. The remainder of the text in the StringItem is line-wrapped to the full width of as many new rows as are necessary to accommodate the text. Each full row is positioned according to the current alignment value. The last line of the text might leave space available on its row. If there is no row break following this StringItem, subsequent Items are packed into the remaining space and the contents of the row are positioned according to the current alignment value. This rule has the effect of displaying the contents of a StringItem as a paragraph of text set flush-left, flush-right, or centered, depending upon
whether the current alignment value is LAYOUT_LEFT, LAYOUT_RIGHT, or LAYOUT_CENTER, respectively. The preferred width and height of a StringItem wrapped across multiple rows, as reported by the \texttt{Item.getPreferredWidth} and \texttt{Item.getPreferredHeight} methods, describe the width and height of the bounding rectangle of the wrapped text.

ImageItems are also treated specially by the above algorithm. The foregoing rules concerning the horizontal alignment value and the \texttt{LAYOUT_LEFT}, \texttt{LAYOUT_RIGHT}, and \texttt{LAYOUT_CENTER} directives, apply to ImageItems only when the \texttt{LAYOUT_2} directive is also present on that item. If the \texttt{LAYOUT_2} directive is not present on an ImageItem, the behavior of the \texttt{LAYOUT_LEFT}, \texttt{LAYOUT_RIGHT}, and \texttt{LAYOUT_CENTER} directives is implementation-specific.

A Form's layout is recomputed automatically as necessary. This may occur because of a change in an Item's size caused by a change in its contents or because of a request by the application to change the Item's preferred size. It may also occur if an Item's layout directives are changed by the application. The application does not need to perform any specific action to cause the Form's layout to be updated.

\section*{Line Breaks and Wrapping}

For all cases where text is wrapped, line breaks must occur at each newline character (\texttt{\textbackslash n} = Unicode \texttt{\textnumero 000A}). If space does not permit the full text to be displayed it is truncated at line breaks. If there are no suitable line breaks, it is recommended that implementations break text at word boundaries. If there are no word boundaries, it is recommended that implementations break text at character boundaries.

Labels that contain line breaks may be truncated at the line break and cause the rest of the label not to be shown.

\section*{User Interaction}

When a Form is present on the display the user can interact with it and its Items indefinitely (for instance, traversing from Item to Item and possibly scrolling). These traversing and scrolling operations do not cause application-visible events. The system notifies the application when the user modifies the state of an interactive Item contained within the Form. This notification is accomplished by calling the \texttt{itemStateChanged()} method of the listener declared to the Form with the \texttt{setItemStateListener()} method.

As with other \texttt{Displayable} objects, a Form can declare \texttt{commands} and declare a command listener with the \texttt{setCommandListener()} method. \texttt{CommandListener} objects are distinct from \texttt{ItemStateListener} objects, and they are declared and invoked separately.

\section*{Notes for Application Developers}
Although this class allows creation of arbitrary combination of components the application developers should keep the small screen size in mind. Form is designed to contain a small number of closely related UI elements.

If the number of items does not fit on the screen, the implementation may choose to make it scrollable or to fold some components so that a separate screen appears when the element is edited.

Since:
MIDP 1.0

See Also:
Item

### Constructor Summary

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Form(String title)</td>
<td>Creates a new, empty Form.</td>
</tr>
<tr>
<td>Form(String title, Item[] items)</td>
<td>Creates a new Form with the specified contents.</td>
</tr>
</tbody>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int append(Image img)</td>
<td>Adds an item consisting of one Image to the Form.</td>
</tr>
<tr>
<td>int append(Item item)</td>
<td>Adds an Item into the Form.</td>
</tr>
<tr>
<td>int append(String str)</td>
<td>Adds an item consisting of one String to the Form.</td>
</tr>
<tr>
<td>void delete(int itemNum)</td>
<td>Deletes the Item referenced by itemNum.</td>
</tr>
<tr>
<td>void deleteAll()</td>
<td>Deletes all the items from this Form, leaving it with zero items.</td>
</tr>
<tr>
<td>Item get(int itemNum)</td>
<td>Gets the item at given position.</td>
</tr>
<tr>
<td>int getHeight()</td>
<td>Returns the height in pixels of the displayable area available for items.</td>
</tr>
<tr>
<td>int getWidth()</td>
<td>Returns the width in pixels of the displayable area available for items.</td>
</tr>
<tr>
<td>void insert(int itemNum, Item item)</td>
<td>Inserts an item into the Form just prior to the item specified.</td>
</tr>
<tr>
<td>void set(int itemNum, Item item)</td>
<td>Sets the item referenced by itemNum to the specified item, replacing the previous item.</td>
</tr>
<tr>
<td>void setItemStateListener(ItemStateListener iListener)</td>
<td>Sets the ItemStateListener for the Form, replacing any previous</td>
</tr>
</tbody>
</table>
ItemStateListener.

```java
public int size()
```

 Gets the number of items in the Form.

**Methods inherited from class javax.microedition.lcdui.Displayable**

`addCommand`, `getTicker`, `getTitle`, `isShown`, `removeCommand`, `setCommandListener`, `setTicker`, `setTitle`, `sizeChanged`

**Methods inherited from class java.lang.Object**

`equals`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString`, `wait`, `wait`, `wait`

**Constructor Detail**

**Form**

```java
public Form(String title)
```

 Creates a new, empty Form.

**Parameters:**

- `title` - the Form's title, or null for no title

**Form**

```java
public Form(String title, Item[] items)
```

 Creates a new Form with the specified contents. This is identical to creating an empty Form and then using a set of append methods. The items array may be null, in which case the Form is created empty. If the items array is non-null, each element must be a valid Item not already contained within another Form.

**Parameters:**

- `title` - the Form's title string
- `items` - the array of items to be placed in the Form, or null if there are no items

**Throws:**

- `IllegalStateException` - if one of the items is already owned by another container
- `NullPointerException` - if an element of the items array is null

**Method Detail**
public int append(Item item)
        Adds an Item into the Form. The newly added Item becomes the last Item in the Form, and the size of the Form grows by one.
        Parameters:
        item - the Item to be added.
        Returns:
        the assigned index of the Item
        Throws:
        IllegalStateException - if the item is already owned by a container
        NullPointerException - if item is null
The `LAYOUT_DEFAULT` directive indicates that the container's default layout policy is to be used for this item. `LAYOUT_DEFAULT` has the value zero and has no effect when combined with other layout directives. It is useful within programs in order to document the programmer's intent.

The `LAYOUT_LEFT`, `LAYOUT_RIGHT`, and `LAYOUT_CENTER` directives indicate horizontal alignment and are mutually exclusive. Similarly, the `LAYOUT_TOP`, `LAYOUT_BOTTOM`, and `LAYOUT_VCENTER` directives indicate vertical alignment and are mutually exclusive.

A horizontal alignment directive, a vertical alignment directive, and any combination of other layout directives may be combined using the bit-wise `OR` operator (`|`) to compose a layout directive value. Such a value is used as the parameter to the `setLayout(int)` method and is the return value from the `getLayout()` method.

Some directives have no defined behavior in some contexts. A layout directive is ignored if its behavior is not defined for the particular context within which the Item resides.

A complete specification of the layout of Items within a Form is given [here](#).

**Item Sizes**

Items have two explicit size concepts: the `minimum` size and the `preferred` size. Both the minimum and the preferred sizes refer to the total area of the Item, which includes space for the Item's contents, the Item's label, as well as other space that is significant to the layout policy. These sizes do not include space that is not significant for layout purposes. For example, if the addition of a label to an Item would cause other Items to move in order to make room, then the space occupied by this label is significant to layout and is counted as part of the Item's minimum and preferred sizes. However, if an implementation were to place the label in a margin area reserved exclusively for labels, this would not affect the layout of neighboring Items. In this case, the space occupied by the label would not be considered part of the minimum and preferred sizes.

The minimum size is the smallest size at which the Item can function and display its contents, though perhaps not optimally. The minimum size may be recomputed whenever the Item's contents changes.
The preferred size is generally a size based on the Item's contents and is the smallest size at which no information is clipped and text wrapping (if any) is kept to a tolerable minimum. The preferred size may be recomputed whenever the Item's contents changes. The application can lock the preferred width or preferred height (or both) by supplying specific values for parameters to the setPreferredSize method. The manner in which an Item fits its contents within an application-specified preferred size is implementation-specific. However, it is recommended that textual content be word-wrapped to fit the preferred size set by the application. The application can unlock either or both dimensions by supplying the value -1 for parameters to the setPreferredSize method.

When an Item is created, both the preferred width and height are unlocked. In this state, the implementation computes the preferred width and height based on the Item's contents, possibly including other relevant factors such as the Item's graphic design and the screen dimensions. After having locked either the preferred width or height, the application can restore the initial, unlocked state by calling setPreferredSize(-1, -1).

The application can lock one dimension of the preferred size and leave the other unlocked. This causes the system to compute an appropriate value for the unlocked dimension based on arranging the contents to fit the locked dimension. If the contents change, the size on the unlocked dimension is recomputed to reflect the new contents, but the size on the locked dimension remains unchanged. For example, if the application called setPreferredSize(50, -1), the preferred width would be locked at 50 pixels and the preferred height would be computed based on the Item's contents. Similarly, if the application called setPreferredSize(-1, 60), the preferred height would be locked at 60 pixels and the preferred width would be computed based on the Item's contents. This feature is particularly useful for Items with textual content that can be line wrapped.

The application can also lock both the preferred width and height to specific values. The Item's contents are truncated or padded as necessary to honor this request. For Items containing text, the text should be wrapped to the specified width, and any truncation should occur at the end of the text.

Items also have an implicit maximum size provided by the implementation. The maximum width is typically based on the width of the screen space available to a Form. Since Forms can scroll vertically, the maximum height should typically not be based on the height of the available screen space.

If the application attempts to lock a preferred size dimension to a value smaller than the minimum or larger than the maximum, the implementation may disregard the requested value and instead use either the minimum or maximum as appropriate. If this occurs, the actual values used must be visible to the application via the values returned from the getPreferredSizeWidth and getPreferredSizeHeight methods.

**Commands**

A Command is said to be present on an Item if the Command has been added to this Item with a prior call to addCommand(javax.microedition.lcdui.Command) or
setDefaultCommand(javax.microedition.lcdui.Command) and if the Command has not been removed with a subsequent call to removeCommand(javax.microedition.lcdui.Command). Commands present on an item should have a command type of ITEM. However, it is not an error for a command whose type is other than ITEM to be added to an item. For purposes of presentation and placement within its user interface, the implementation is allowed to treat a command's items as if they were of type ITEM.

Items may have a default Command. This state is controlled by the setDefaultCommand(javax.microedition.lcdui.Command) method. The default Command is eligible to be bound to a special platform-dependent user gesture. The implementation chooses which gesture is the most appropriate to initiate the default command on that particular Item. For example, on a device that has a dedicated selection key, pressing this key might invoke the item's default command. Or, on a stylus-based device, tapping on the Item might invoke its default command. Even if it can be invoked through a special gesture, the default command should also be invokable in the same fashion as other item commands.

It is possible that on some devices there is no special gesture suitable for invoking the default command on an item. In this case the default command must be accessible to the user in the same fashion as other item commands. The implementation may use the state of a command being the default in deciding where to place the command in its user interface.

It is possible for an Item not to have a default command. In this case, the implementation may bind its special user gesture (if any) for another purpose, such as for displaying a menu of commands. The default state of an Item is not to have a default command. An Item may be set to have no default Command by removing it from the Item or by passing null to the setDefaultCommand() method.

The same command may occur on more than one Item and also on more than one Displayable. If this situation occurs, the user must be provided with distinct gestures to invoke that command on each Item or Displayable on which it occurs, while those Items or Displayables are visible on the display. When the user invokes the command, the listener (CommandListener or ItemCommandListener as appropriate) of just the object on which the command was invoked will be called.

Adding commands to an Item may affect its appearance, the way it is laid out, and the traversal behavior. For example, the presence of commands on an Item may cause row breaks to occur, or it may cause additional graphical elements (such as a menu icon) to appear. In particular, if a StringItem whose appearance mode is PLAIN (see below) is given one or more Commands, the implementation is allowed to treat it as if it had a different appearance mode.

Appearance Modes

The StringItem and ImageItem classes have an appearance mode attribute that can be set in their constructors. This attribute can have one of the values PLAIN, HYPERLINK, or BUTTON. An appearance mode of PLAIN is typically used for non-interactive display of
textual or graphical material. The appearance mode values do not have any side effects on the interactivity of the item. In order to be interactive, the item must have one or more Commands (preferably with a default command assigned), and it must have a CommandListener that receives notification of Command invocations. The appearance mode values also do not have any effect on the semantics of Command invocation on the item. For example, setting the appearance mode of a StringItem to be HYPERLINK requests that the implementation display the string contents as if they were a hyperlink in a browser. It is the application's responsibility to attach a Command and a listener to the StringItem that provide behaviors that the user would expect from invoking an operation on a hyperlink, such as loading the referent of the link or adding the link to the user's set of bookmarks.

Setting the appearance mode of an Item to be other than PLAIN may affect its minimum, preferred, and maximum sizes, as well as the way it is laid out. For example, a StringItem with an appearance mode of BUTTON should not be wrapped across rows. (However, a StringItem with an appearance mode of HYPERLINK should be wrapped the same way as if its appearance mode is PLAIN.)

A StringItem or ImageItem in BUTTON mode can be used to create a button-based user interface. This can easily lead to applications that are inconvenient to use. For example, in a traversal-based system, users must navigate to a button before they can invoke any commands on it. If buttons are spread across a long Form, users may be required to perform a considerable amount of navigation in order to discover all the available commands. Furthermore, invoking a command from a button at the other end of the Form can be quite cumbersome. Traversal-based systems often provide a means of invoking commands from anywhere (such as from a menu), without the need to traverse to a particular item. Instead of adding a command to a button and placing that button into a Form, it would often be more appropriate and convenient for users if that command were added directly to the Form. Buttons should be used only in cases where direct user interaction with the item's string or image contents is essential to the user's understanding of the commands that can be invoked from that item.

**Default State**

Unless otherwise specified by a subclass, the default state of newly created Items is as follows:

- the Item is not contained within ("owned by") any container;
- there are no Commands present;
- the default Command is null;
- the ItemCommandListener is null;
- the layout directive value is LAYOUT_DEFAULT; and
- both the preferred width and preferred height are unlocked.

**Since:**

MIDP 1.0

---

**Field Summary**
<table>
<thead>
<tr>
<th>Static Integer</th>
<th>Appearance Mode/Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BUTTON</strong></td>
<td>An appearance mode value indicating that the Item is to appear as a button.</td>
</tr>
<tr>
<td><strong>HYPERLINK</strong></td>
<td>An appearance mode value indicating that the Item is to appear as a hyperlink.</td>
</tr>
<tr>
<td><strong>LAYOUT_2</strong></td>
<td>A layout directive indicating that new MIDP 2.0 layout rules are in effect for this Item.</td>
</tr>
<tr>
<td><strong>LAYOUT_BOTTOM</strong></td>
<td>A layout directive indicating that this Item should have a bottom-aligned layout.</td>
</tr>
<tr>
<td><strong>LAYOUT_CENTER</strong></td>
<td>A layout directive indicating that this Item should have a horizontally centered layout.</td>
</tr>
<tr>
<td><strong>LAYOUT_DEFAULT</strong></td>
<td>A layout directive indicating that this Item should follow the default layout policy of its container.</td>
</tr>
<tr>
<td><strong>LAYOUT_EXPAND</strong></td>
<td>A layout directive indicating that this Item's width may be increased to fill available space.</td>
</tr>
<tr>
<td><strong>LAYOUT_LEFT</strong></td>
<td>A layout directive indicating that this Item should have a left-aligned layout.</td>
</tr>
<tr>
<td><strong>LAYOUT_NEWLINE_AFTER</strong></td>
<td>A layout directive indicating that this Item should be the last on its line or row, and that the next Item (if any) in the container should be placed on a new line or row.</td>
</tr>
<tr>
<td><strong>LAYOUT_NEWLINE_BEFORE</strong></td>
<td>A layout directive indicating that this Item should be placed at the beginning of a new line or row.</td>
</tr>
<tr>
<td><strong>LAYOUT_RIGHT</strong></td>
<td>A layout directive indicating that this Item should have a right-aligned layout.</td>
</tr>
<tr>
<td><strong>LAYOUT_SHRINK</strong></td>
<td>A layout directive indicating that this Item's width may be reduced to its minimum width.</td>
</tr>
<tr>
<td><strong>LAYOUT_TOP</strong></td>
<td>A layout directive indicating that this Item should have a top-aligned layout.</td>
</tr>
<tr>
<td><strong>LAYOUT_VCENTER</strong></td>
<td>A layout directive indicating that this Item should have a vertically centered layout.</td>
</tr>
</tbody>
</table>
| **LAYOUT_VEXPAND** | A layout directive indicating that this Item's height may be
increased to fill available space.

<table>
<thead>
<tr>
<th>static int</th>
<th>LAYOUT_VSHRINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>A layout directive indicating that this Item's height may be reduced to its minimum height.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>static int</th>
<th>PLAIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>An appearance mode value indicating that the Item is to have a normal appearance.</td>
<td></td>
</tr>
</tbody>
</table>

### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void <strong>addCommand</strong>(Command cmd)</td>
<td>Adds a context sensitive Command to the item.</td>
</tr>
<tr>
<td>String <strong>getLabel</strong>()</td>
<td>Gets the label of this Item object.</td>
</tr>
<tr>
<td>int <strong>getLayout</strong>()</td>
<td>Gets the layout directives used for placing the item.</td>
</tr>
<tr>
<td>int <strong>getMinimumHeight</strong>()</td>
<td>Gets the minimum height for this Item.</td>
</tr>
<tr>
<td>int <strong>getMinimumWidth</strong>()</td>
<td>Gets the minimum width for this Item.</td>
</tr>
<tr>
<td>int <strong>getPreferredHeight</strong>()</td>
<td>Gets the preferred height of this Item.</td>
</tr>
<tr>
<td>int <strong>getPreferredWidth</strong>()</td>
<td>Gets the preferred width of this Item.</td>
</tr>
<tr>
<td>void <strong>notifyStateChanged</strong>()</td>
<td>Causes this Item's containing Form to notify the Item's ItemStateListener.</td>
</tr>
<tr>
<td>void <strong>removeCommand</strong>(Command cmd)</td>
<td>Removes the context sensitive command from item.</td>
</tr>
<tr>
<td>void <strong>setDefaultCommand</strong>(Command cmd)</td>
<td>Sets default Command for this Item.</td>
</tr>
<tr>
<td>void <strong>setItemCommandListener</strong>(ItemCommandListener l)</td>
<td>Sets a listener for Commands to this Item, replacing any previous ItemCommandListener.</td>
</tr>
<tr>
<td>void <strong>setLabel</strong>(String label)</td>
<td>Sets the label of the Item.</td>
</tr>
<tr>
<td>void <strong>setLayout</strong>(int layout)</td>
<td>Sets the layout directives for this item.</td>
</tr>
<tr>
<td>void <strong>setPreferredSize</strong>(int width, int height)</td>
<td>Sets the preferred width and height for this Item.</td>
</tr>
</tbody>
</table>